Towards Requirements-Driven Information Systems Engineering: The *Tropos* Project

Jaelson Castro [a] Manuel Kolp [b,1] John Mylopoulos [c]

[a] *Universidade Federal de Pernambuco, Centro de Informática, Av. Prof. Luiz Freire S/N, Recife PE, Brazil 50732-970*

[b] *University of Louvain, IAG School of Management, Information Systems Research Unit, 1 Place des Doyens, B-1348, Louvain-La-Neuve, Belgium*

[c] *University of Toronto, Department of Computer Science, 6 King's College Road, Toronto M5S 3H5, Ontario, Canada*

Abstract

Information systems of the future will have to perform well within ever-changing organizational environments. Unfortunately, existing software development methodologies (object-oriented, structured or otherwise) have traditionally been inspired by programming concepts, not organizational ones, leading to a semantic gap between the software system and its operational environment. To reduce this gap, we propose a software development methodology named *Tropos* which is founded on concepts used to model early requirements. Our proposal adopts the *i** organizational modeling framework, which offers the notions of *actor*, *goal* and (actor) *dependency*, and uses these as a foundation to model early and late requirements, architectural and detailed design. The paper outlines *Tropos* phases through an e-business example, and sketches a formal language which underlies the methodology and is intended to support formal analysis. The methodology seems to complement well proposals for agent-oriented programming platforms.

Key words:
Software development methodology, requirements engineering, information systems analysis and design, agent-oriented systems, software architectures.

[1] Corresponding author. Tel.: +32-10-47-8395; fax: +32-10-47-8324; e-mail: kolp@isys.ucl.ac.be

1 Introduction

Information systems have traditionally suffered from an impedance mismatch. Their operational environment is understood in terms of actors, responsibilities, objectives, tasks and resources, while the information system itself is conceived as a collection of (software) modules, entities (e.g., objects, agents), data structures and interfaces. This mismatch is one of the main factors for the poor quality of information systems, also the frequent failure of system development projects.

One cause of this mismatch is that development methodologies have traditionally been inspired and driven by the programming paradigm of the day. This means that the concepts, methods and tools used during all phases of development were based on those offered by the pre-eminent programming paradigm. So, during the era of structured programming, structured analysis and design techniques were proposed [15,49], while object-oriented programming has given rise more recently to object-oriented design and analysis [4,45]. For structured development techniques this meant that throughout software development, the developer could conceptualize the system in terms of functions and processes, inputs and outputs. For object-oriented development, on the other hand, the developer thinks throughout in terms of objects, classes, methods, inheritance and the like.

Using the same concepts to align requirements analysis with system design and implementation makes perfect sense. For one thing, such an alignment reduces impedance mismatches between different development phases. Moreover, such an alignment can lead to coherent toolsets and techniques for developing system (and it has!) as well, it can streamline the development process itself.

But, why base such an alignment on implementation concepts? Requirements analysis is arguably the most important stage of information system development. This is the phase where technical considerations have to be balanced against social and organizational ones and where the operational environment of the system is modeled. Not surprisingly, this is also the phase where the most and costliest errors are introduced to a system. Even if (or rather, when) the importance of design and implementation phases wanes sometime in the future, requirements analysis will remain a critical phase for the development of any information system, answering the most fundamental of all design questions: "what is the system intended for?"

Information systems of the future, such as enterprise resource planning (ERP), groupware, knowledge management and e-business systems, should be designed to match their operational environment. For instance, ERP systems have to implement a process view of the enterprise to meet business goals,

tightly integrating all relevant functions of their operational environment. To reduce as much as possible this impedance mismatch between the system and its environment, we outline in this paper a development framework, named *Tropos* [2], which is requirements-driven in the sense that it is based on concepts used during early requirements analysis. To this end, we adopt the concepts offered by i^* [52], a modeling framework proposing concepts such as *actor* (actors can be *agents*, *positions* or *roles*), as well as social dependencies among actors, including *goal*, *softgoal*, *task* and *resource* dependencies. These concepts are used for an e-commerce example [3] to model not just early requirements, but also late requirements, as well as architectural and detailed design. The proposed methodology spans four phases:

- Early requirements, concerned with the understanding of a problem by studying an organizational setting; the output of this phase is an organizational model which includes relevant actors, their respective goals and their inter-dependencies.
- Late requirements, where the system-to-be is described within its operational environment, along with relevant functions and qualities.
- Architectural design, where the system's global architecture is defined in terms of subsystems, interconnected through data, control and other dependencies.
- Detailed design, where behaviour of each architectural component is defined in further detail.

The proposed methodology includes techniques for generating an implementation from a *Tropos* detailed design. Using an agent-oriented programming platform for the implementation seems natural, given that the detailed design is defined in terms of (system) actors, goals and inter-dependencies among them. For this paper, we have adopted JACK as programming platform to study the generation of an implementation from a detailed design. JACK is a commercial product based on the BDI (Beliefs-Desires-Intentions) agent architecture.

This paper is an extended and revised version of [7] and integrates further results from [6,20,21,32–34,37]. Section 2 of the paper describes a case study for a B2C (business to consumer) e-commerce application. Section 3 introduces the primitive concepts offered by i^* and illustrates their use with an example. Sections 4, 5, and 6 illustrate how the technique works for late requirements, architectural design and detailed design respectively. Section 7 sketches the

[2] For further detail and information about the *Tropos* project, see http://www.cs.toronto.edu/km/tropos.

[3] Although, we could have included a simpler (toy) example, we decided to present a more realistic e-commerce system development exercise of moderate complexity [12].

implementation of the case study using the JACK agent development environment. Finally, Section 8 summarizes the contributions of the paper and relates it to the literature while Appendix A summarizes the methodology.

2 A Case Study

Media Shop is a store selling and shipping different kinds of media items such as books, newspapers, magazines, audio CDs, videotapes, and the like. *Media Shop* customers (on-site or remote) can use a periodically updated catalogue describing available media items to specify their order. *Media Shop* is supplied with the latest releases from *Media Producer* and in-catalogue items by *Media Supplier*. To increase market share, *Media Shop* has decided to open up a B2C retail sales front on the internet. With the new setup, a customer can order *Media Shop* items in person, by phone, or through the internet. The system has been named *Medi@* and is available on the world-wide-web using communication facilities provided by *Telecom Cpy*. It also uses financial services supplied by *Bank Cpy*, which specializes on on-line transactions. The basic objective for the new system is to allow an on-line customer to examine the items in the *Medi@* internet catalogue, and place orders.

There are no registration restrictions, or identification procedures for *Medi@* users. Potential customers can search the on-line store by either browsing the catalogue or querying the item database. The catalogue groups media items of the same type into (sub)hierarchies and genres (e.g., audio CDs are classified into pop, rock, jazz, opera, world, classical music, soundtrack, ...) so that customers can browse only (sub)categories of interest. An on-line search engine allows customers with particular items in mind to search title, author/artist and description fields through keywords or full-text search. If the item is not available in the catalogue, the customer has the option of asking *Media Shop* to order it, provided the customer has editor/publisher references (e.g., ISBN, ISSN), and identifies herself (in terms of name and credit card number). Details about media items include title, media category (e.g., book) and genre (e.g., science-fiction), author/artist, short description, editor/publisher international references and information, date, cost, and sometimes pictures (when available).

3 Early Requirements Analysis with *i**

Early requirements analysis focuses on the intentions of stakeholders. These intentions are modeled as goals which, through some form of a goal-oriented analysis, eventually lead to the functional and non-functional requirements of

the system-to-be [13]. In *i** (which stands for "distributed intentionality"), stakeholders are represented as (social) actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The *i** framework includes the *strategic dependency model* for describing the network of relationships among actors, as well as the *strategic rationale model* for describing and supporting the reasoning that each actor goes through concerning its relationships with other actors. These models have been formalized using intentional concepts from Artificial Intelligence, such as goal, belief, ability, and commitment (e.g., [11]). The framework has been presented in detail in [24,52] and has been related to different application areas, including requirements engineering [50], software processes [51], and business process reengineering [53].

A strategic dependency model is a graph involving *actors* who have *strategic dependencies* among each other. A dependency describes an "agreement" (called *dependum*) between two actors: the *depender* and the *dependee*. The *depender* is the depending actor, and the *dependee,* the actor who is depended upon. The type of the dependency describes the nature of the agreement. *Goal* dependencies are used to represent delegation of responsibility for fulfilling a goal; *softgoal* dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely (for instance, the appreciation is subjective, or the fulfillment can occur only to a given extent); *task* dependencies are used in situations where the dependee is required to perform a given activity; and *resource* dependencies require the dependee to provide a resource to the depender. As shown in Figure 1, actors are represented as circles; dependums – goals, softgoals, tasks and resources – are respectively represented as ovals, clouds, hexagons and rectangles; and dependencies have the form *depender* → *dependum* → *dependee*.

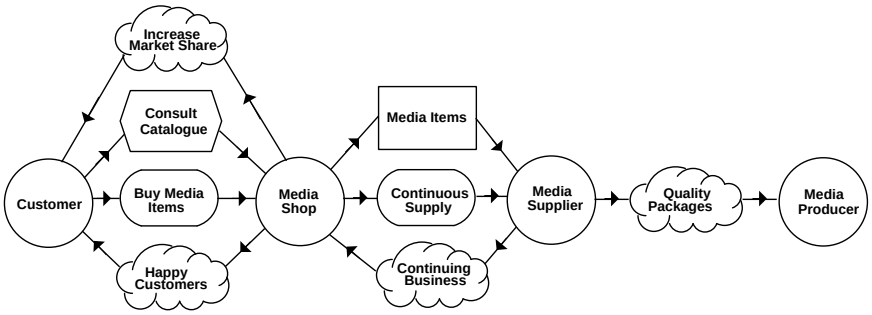

Fig. 1. *i** Model for a Media Shop

These elements are sufficient for producing a first model of an organizational environment. For instance, Figure 1 depicts an *i** model of our *Medi@* example. The main actors are *Customer*, *Media Shop*, *Media Supplier* and *Media Producer*. *Customer* depends on *Media Shop* to fulfill her goal: *Buy Media*

Items. Conversely, *Media Shop* depends on *Customer* to *increase market share* and make "*customers happy*". Since the dependum *Happy Customers* cannot be defined precisely, it is represented as a softgoal. The *Customer* also depends on *Media Shop* to *consult the catalogue* (task dependency). Furthermore, *Media Shop* depends on *Media Supplier* to supply media items in a continuous way and get a *Media Item* (resource dependency). The items are expected to be of good quality because, otherwise, the *Continuing Business* dependency would not be fulfilled. Finally, *Media Producer* is expected to provide *Media Supplier* with *Quality Packages.*

We have defined a formal language, called *Formal Tropos* [21], that complements i^* in several directions. First of all, it provides a textual notation for i^* models and allows us to describe dynamic constraints among the different elements of a specification in a first order, linear-time temporal logic. Second, it has a precisely defined semantics that is amenable to formal analysis. Finally, *Formal Tropos* comes with a methodology for the automated analysis and animation of specifications [21], based on model checking techniques [9].

As an example, Figure 2 presents the specification for the *Buy Media Items* and *Continuous Supply* goal dependencies. Notice that this specification provides additional information not present in the i^* diagram. For instance, the *fulfillment condition* of *Buy Media Items* states that the customer expects to get the best price for the type of product she is buying. The condition for *Continuous Supply* states that the shop expects to have the items in stock as soon as someone is interested in buying them.

Entity Media Item
  Attribute constant type : Type, price : Amount, inStock : Boolean

Dependency Buy Media Items
  Type goal
  Mode achieve
  Depender Customer
  Dependee Media Shop
  Attribute constant item : Media Item
  Fulfillment
    condition for depender
      $\forall\, media : MediaItem(self.item.type = media.type \rightarrow item.price \leq media.price)$
    [the customer expects to get the best price for the type of item]

Dependency Continuous Supply
  Type goal
  Mode maintain
  Depender Media Shop
  Dependee Media Supplier
  Attribute constant item : Media Item
  Fulfillment
    condition for depender
      $\exists buy : BuyItem(JustCreated(buy) \rightarrow buy.item.inStock)$
    [the media retailer expects to get items in stock as soon as
    someone is interested in buying them]

Fig. 2. *A Formal Tropos* Specification

Once the relevant stakeholders and their goals have been identified, a strategic rationale model determines through a means-ends analysis how these goals (including softgoals) can actually be fulfilled through the contributions of other actors. A strategic rationale model is a graph with four types of nodes - *goal*, *task*, *resource*, and *softgoal* - and two types of links - means-ends links and task decomposition links. A strategic rationale graph captures the relationship between the goals of each actor and the dependencies through which the actor expects these dependencies to be fulfilled.

Figure 3 focuses on one of the (soft)goal dependency identified for *Media Shop,* namely *Increase Market Share*. To achieve that softgoal, the analysis postulates a goal *Run Shop* that can be fulfilled by means of a task *Run Shop*. Tasks are partially ordered sequences of steps intended to accomplish some (soft)goal. Tasks can be decomposed into goals and/or subtasks, whose collective fulfillment completes the task. In the figure, *Run Shop* is decomposed into goals *Handle Billing* and *Handle Customer Orders*, tasks *Manage Staff* and *Manage Inventory*, and softgoal *Improve Service* which together accomplish the top-level task. Sub-goals and subtasks can be specified more precisely through refinement. For instance, the goal *Handle Customer Orders* is fulfilled either through tasks *Order By Phone, Order In Person* or *Order By Internet* while the task *Manage Inventory* would be collectively accomplished by tasks *Sell Stock* and *Enhance Catalogue*. These decompositions eventually allow us to identify actors who can accomplish a goal, carry out a task, or deliver on some needed resource for *Media Shop*. Such dependencies in Figure 3 are, among others, the goal and resource dependencies on *Media Supplier* for supplying, in a continuous way, media items to enhance the catalogue and sell products, the softgoal dependencies on *Customer* for increasing market share (by running the shop) and making customers happy (by improving service), and the task dependency *Accouting* on *Bank Cpy* to keep track of business transactions.

4 Late Requirements Analysis

Late requirements analysis results in a requirements specification which describes all functional and non-functional requirements for the system-to-be. In *Tropos*, the information system is represented as one or more actors which participate in a strategic dependency model, along with other actors from the system's operational environment. In other words, the system comes into the picture as one or more actors who contribute to the fulfillment of stakeholder goals.

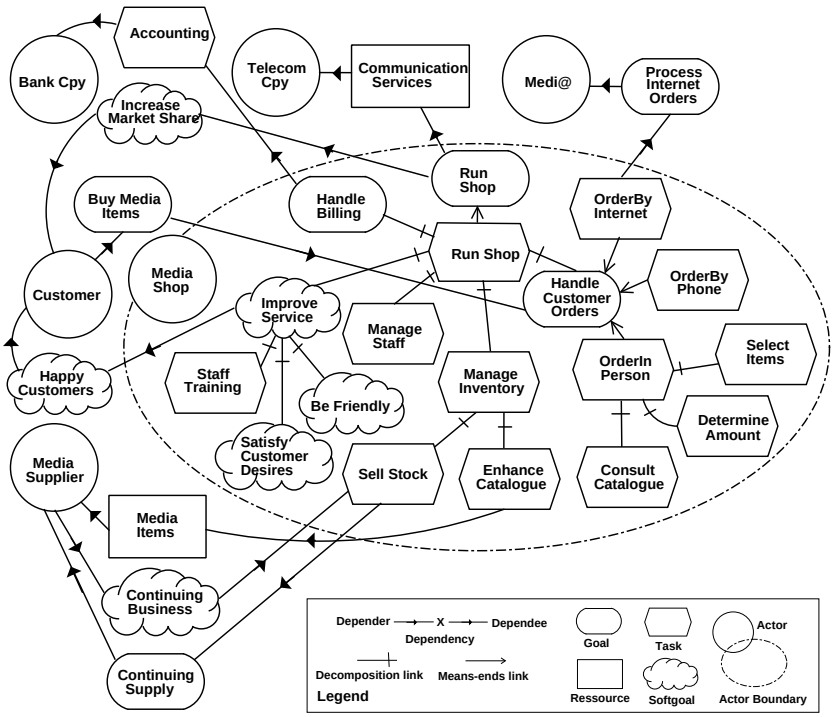

Fig. 3. Means-Ends Analysis for the Softgoal *Increase Market Share*

For our example, the *Medi@* system is introduced as an actor in the strategic dependency model depicted in Figure 4. With respect to the actors previously identified, *Customer* depends on *Media Shop* to buy media items while *Media Shop* depends on *Customer* to increase market share and make them happy (with *Media Shop* service). *Media Supplier* is expected to supply *Media Shop* with media items in a continuous way since depending on the latter for continuing business. It can also use *Medi@* to determine new needs from customers, such as media items not available in the catalogue while expecting *Media Producer* to provide her with *quality packages*. As indicated earlier, *Media Shop* depends on *Medi@* for processing internet orders and on *Bank Cpy* to process business transactions. *Customer*, in turn, depends on *Medi@* to place orders through the internet, to search the database for keywords, or simply to browse the on-line catalogue. With respect to relevant qualities, *Customer* requires that transaction services be secure and usable, while *Media Shop* expects *Medi@* to be easily adaptable (e.g., catalogue enhancing, item database evolution, user interface update, ...). Finally, *Medi@* relies on internet services provided by *Telecom Cpy* and on secure on-line financial transactions handled by *Bank Cpy*.

Although a strategic dependency model provides hints about why processes are structured in a certain way, it does not sufficiently support the process of suggesting, exploring, and evaluating alternative solutions. As late require-

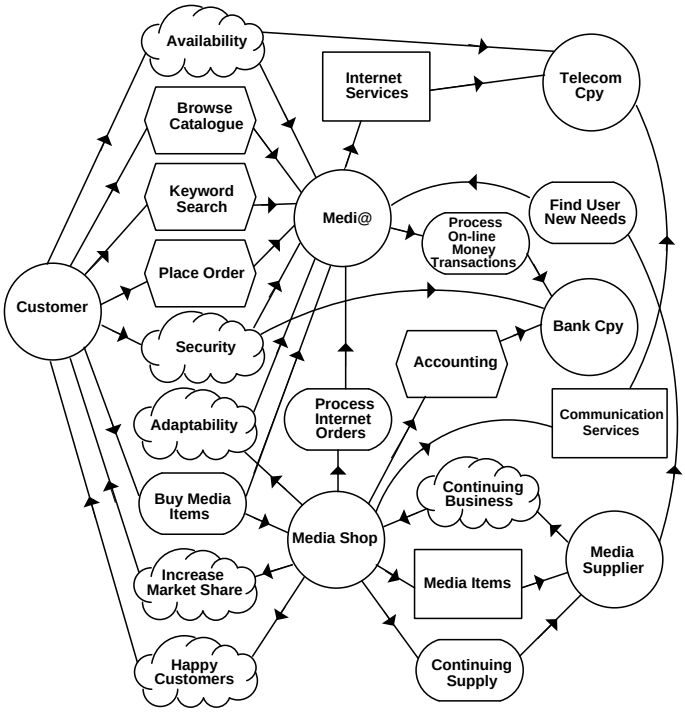

Fig. 4. Strategic Dependency Model for a Media Shop

ments analysis proceeds, *Medi@* is given additional responsibilities, and ends up as the depender of several dependencies. Moreover, the system is decomposed into several sub-actors which take on some of these responsibilities. This decomposition and responsibility assignment is realized using the same kind of means-ends analysis along with the strategic rationale analysis illustrated in Figure 3. Hence, the analysis in Figure 5 focuses on the system itself, instead of an external stakeholder.

The figure postulates a root task *Internet Shop Managed* providing sufficient support (++) [8] to the softgoal *Increase Market Share*. That task is firstly refined into goals *Internet Order Handled* and *Item Searching Handled*, softgoals *Attract New Customer*, *Secure* and *Available*, and tasks *Produce Statistics* and *Adaptation*. To manage internet orders, *Internet Order Handled* is achieved through the task *Shopping Cart* which is decomposed into subtasks *Select Item*, *Add Item*, *Check Out,* and *Get Identification Detail*. These are the main process activities required to design an operational on-line shopping cart [12]. The latter (task) is achieved either through sub-goal *Classic Communication Handled* dealing with phone and fax orders or *Internet Handled* managing secure or standard form orderings. To allow for the ordering of new items not listed in the catalogue, *Select Item* is also further refined into two alternative subtasks, one dedicated to select catalogued items, the other to preorder unavailable products. To provide sufficient support (++) to the *Adaptable* softgoal, *Adaptation* is refined into four subtasks dealing with catalogue updates, system evolution, interface updates and system monitor-

ing. The goal *Item Searching Handled* might alternatively be fulfilled through tasks *Database Querying* or *Catalogue Consulting* with respect to customers' navigating desiderata, i.e., searching with particular items in mind by using search functions or simply browsing the catalogued products.

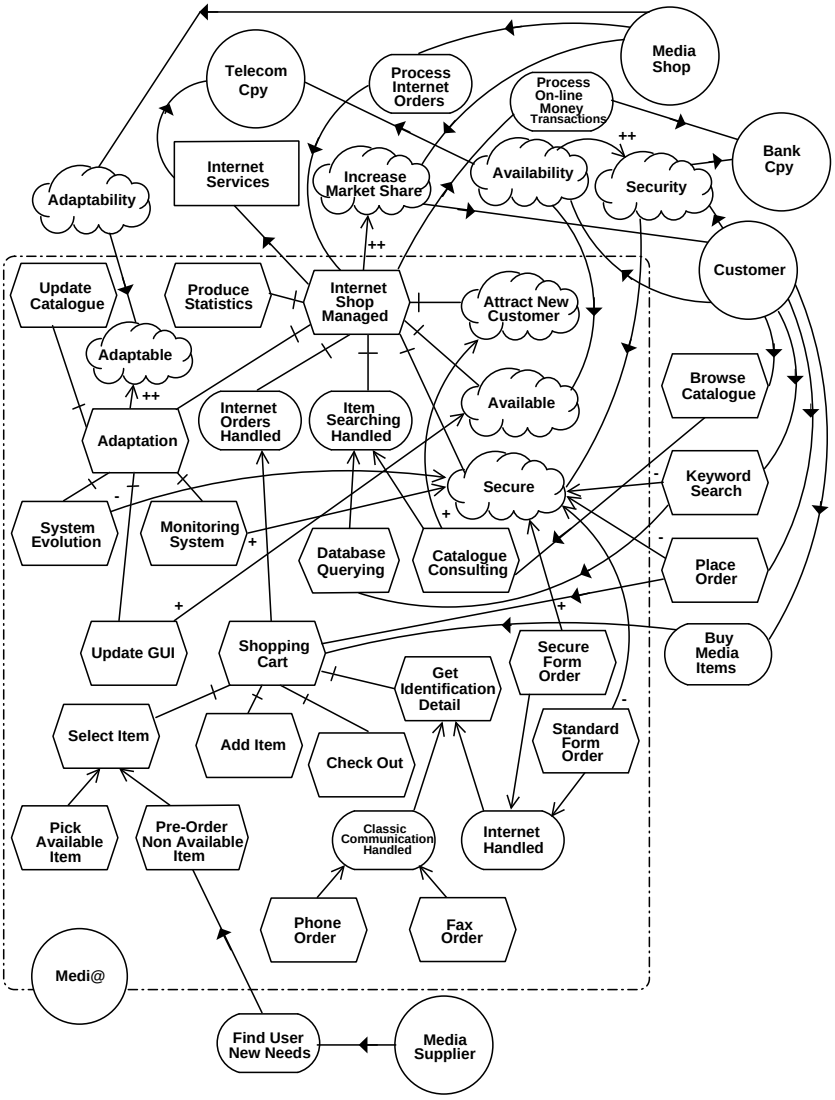

Fig. 5. Strategic Rationale Model for *Medi@*

In addition, as already pointed, Figure 5 introduces softgoal contributions to model sufficient/partial positive (respectively ++ and +) or negative (respectively −− and −) support to softgoals *Secure*, *Available*, *Adaptable*, *Attract New Customers* and *Increase Market Share*. The result of this means-ends analysis is a set of (system and human) actors who are dependees for some of the dependencies that have been postulated.

Resource, task and softgoal dependencies correspond naturally to functional

and non-functional requirements. Leaving (some) goal dependencies between system actors and other actors is a novelty. Traditionally, functional goals are "operationalized" during late requirements [13], while quality softgoals are either operationalized or "metricized" [14]. For example, *Billing Processor* may be operationalized during late requirements analysis into particular business processes for processing bills and orders. Likewise, a security softgoal might be operationalized by defining interfaces which minimize input/output between the system and its environment, or by limiting access to sensitive information. Alternatively, the security requirement may be metricized into something like "No more than X unauthorized operations in the system-to-be per year".

Leaving goal dependencies with system actors as dependees makes sense whenever there is a foreseeable need for flexibility in the performance of a task on the part of the system. For example, consider a communication goal "communicate X to Y". According to conventional development techniques, such a goal needs to be operationalized before the end of late requirements analysis, perhaps into some sort of a user interface through which user Y will receive message X from the system. The problem with this approach is that the steps through which this goal is to be fulfilled (along with a host of background assumptions) are frozen into the requirements of the system-to-be. This early translation of goals into concrete plans for their fulfillment makes systems fragile and less reusable.

In our example, we have left three (soft)goals (*Availability, Security, Adaptability*) in the late requirements model. The first goal is *Availability* because we propose to allow system agents to automatically decide at run-time which catalogue browser, shopping cart and order processor architecture fit best customer needs or navigator/platform specifications. Moreover, we would like to include different search engines, reflecting different search techniques, and let the system dynamically choose the most appropriate. The second key softgoal in the late requirements specification is *Security*. To fulfil it, we propose to support in the system's architecture a number of security strategies and let the system decide at run-time which one is the most appropriate, taking into account environment configurations, web browser specifications and network protocols used. The third goal is *Adaptability,* meaning that catalogue content, database schema, and architectural model can be dynamically extended or modified to integrate new and future web-related technologies.

5 Architectural Design

A system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together. By now, in addition to classical architectural styles (e.g., [43]), software

architects have developed catalogues of style for e-business applications [12,26] such as *Thin Web Client*, *Thick Web Client*, *Web Delivery*. Unfortunately, these architectural styles focus on web concepts, protocols and underlying technologies but not on business processes nor non functional requirements of the application. As a result, the organizational architecture styles are not described nor the conceptual high-level perspective of the e-business application. In *Tropos*, we have defined organizational architectural styles [20,32–34] for cooperative, dynamic and distributed applications like mutli-agent systems to guide the design of the system architecture.

These architectural styles (*flat structure, pyramid, joint venture, structure-in-5, takeover, arm's length, vertical integration, co-optation, bidding, ...*) are based on concepts and design alternatives coming from research in organization management: organization theory (e.g.,[42]), strategic alliances and partnerships (e.g.,[17]), theory of the firm (e.g.,[29]), agency theory (e.g.,[2]), ...

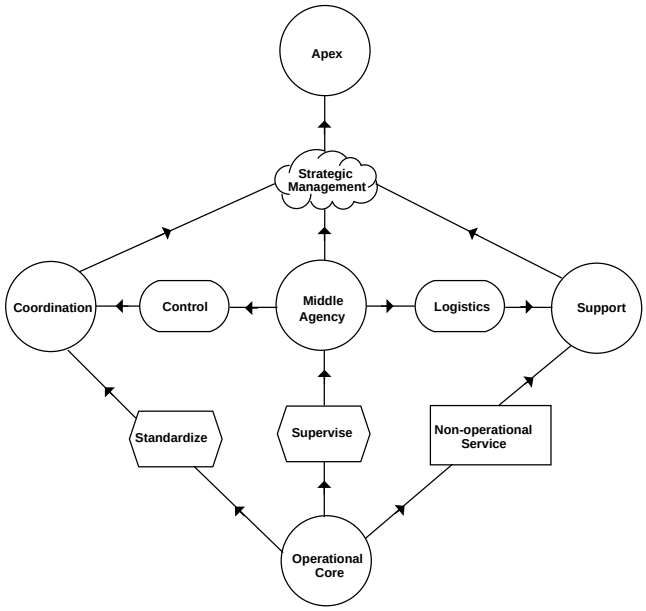

Fig. 6. Structure-in-5

For instance, the *structure-in-5* (Figure 6) is a typical organizational style. At the base level, the *Operational Core* takes care of the basic tasks – the input, processing, output and direct support procedures – associated with running the organization. At the top lies the *Apex*, composed of strategic executive actors. Below it, sit the *Coordination*, *Middle Agency* and *Support* actors, who are in charge of control/standardization, management and logistics procedures, respectively. The *Coordination* component carries out the tasks of standardizing the behavior of other components, in addition to applying analytical procedures to help the organization adapt to its environment. Actors joining the apex to the operational core make up the *Middle Agency*. The

Support component assists the operational core for non-operational services that are outside the basic flow of operational tasks and procedures.

The *joint venture* style (Figure 7) is a more decentralized style based on an agreement between two or more principal partners who benefit from operating at a larger scale and reuse the experience and knowledge of their partners. Each principal partner is autonomous on a local dimension and interacts directly with other principal partners to exchange services, data and knowledge. However, the strategic operation and coordination of the joint venture is delegated to a *Joint Management* actor, who coordinates tasks and manages the sharing of knowledge and resources. Outside the joint venture, secondary partners supply services or support tasks for the organization core.

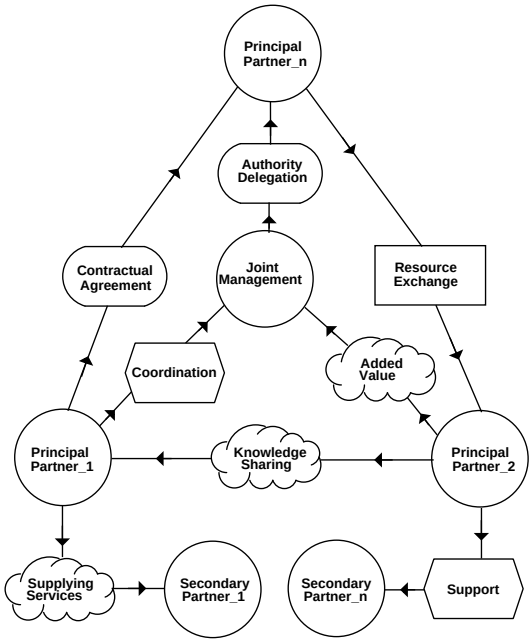

Fig. 7. Joint Venture

The organizational styles are generic structures defined at a metalevel that can be instantiated to design a specific application architecture (see Figure 9 for the joint venture style).

For instance, the structure-in-5 style is a metaclass, *StructureIn5MetaClass*, aggregation of five (*part*) metaclasses, one for each actor composing the structure-in-5 style: *ApexMetaClass*, *CoordinationMetaClass*, *MiddleAgencyMetaClass*, *SupportMetaClass* and *OperationalCoreMetaClass*. Each of these five components exclusively belongs to the composite (*StructureIn5MetaClass*), and their existence depends on the existence of the composite. We are working on the formalization of these styles in *Formal Tropos* [23].

These organizational styles have been evaluated and compared using software quality attributes identified for architectures involving coordinated autonomous components (e.g., Web, internet, agent or peer-to-peer software systems) such as *predictability* (**1**), *security* (**2**), *adaptability* (**3**), *coordinability* (**4**), *cooperativity* (**5**), *availability* (**6**), *integrity* (**7**), *modularity* (**8**), or *aggregability* (**9**). Table 1 summarizes the correlation catalogue for the organizational styles and quality attributes considered in [32,33]. Following notations used by the NFR (non functional requirements) framework [8], +, ++, –, – – respectively model partial/positive, sufficient/positive, partial/negative and sufficient/negative contributions.

Table 1

Correlation Catalogue

	1	2	3	4	5	6	7	8	9
Flat Structure	– –	– –	–			+	+	++	–
Structure-in-5	+	+		+	–	+	++	++	++
Pyramid	++	++	+	++	–	+	– –	–	
Joint-Venture	+	+	++	+	–	++		+	++
Bidding	– –	– –	++	–	++	–	– –	++	
Takeover	++	++	-	++	– –	+		+	+
Arm's-Length	–	– –	+	–	++	– –	++	+	
Hierchical Contracting			+	+	+	+		+	+
Vertical Integration	+	+	–	+	–	+	– –	– –	– –
Cooptation	–	–	++	++	+	– –	–	– –	

The first task during architectural design is to select among alternative architectural styles using as criteria the desired qualities identified earlier. In *Tropos*, we use the NFR framework [8] to conduct such quality analysis. The analysis involves refining these qualities, represented as softgoals, to sub-goals that are more specific and more precise and then evaluating alternative architectural styles against them, as shown in Figure 8. The styles are represented as operationalized softgoals (saying, roughly, "make the architecture of the new system *pyramid-/joint venture-/co-optation*-based, ..."). Design rationale is represented by claim softgoals drawn as dashed clouds. These can represent contextual information (such as priorities) to be considered and properly reflected into the decision making process. Exclamation marks (! and !!) are used to mark priority softgoals. A check-mark "$\surd$" indicates a fulfilled softgoal, while a cross "$\times$" labels an unfulfillable one.

Software quality attributes *Security*, *Availability* and *Adaptability* have been left in the late requirements model (See Section 4). They will guide the selection process of the appropriate architectural style.

In Figure 8, *Adaptability* is AND-decomposed into *Dynamicity* and *Updatability*. For our e-commerce example, *dynamicity* should deal with the way the system can be designed using generic mechanisms to allow web pages and user interfaces to be dynamically and easily changed. Indeed, information content and layout need to be frequently refreshed to give correct information to customers or simply be fashionable for marketing reasons. Frameworks like Active Server Pages (ASP), Server Side Includes (SSI) to create dynamic pages make this attribute easier to achieve. *Updatability* should be strategically important for the viability of the application, the stock management and the business itself since *Media Shop* employees have to very regularly bring up to date the catalogue for inventory consistency.

Availability is decomposed into *Usability*, *Integrity* and *Response Time*. Network communication may not be very reliable causing sporadic loss of the server. There should be data integrity concerns with the capability of the e-business system to do what needs to be done, as quickly and efficiently as possible: in particular with the ability of the system to respond in time to client requests for its services. It is also important to provide the customer with a usable application, i.e., comprehensible at first glimpse, intuitive and ergonomic. Equally strategic to usability concerns is the portability of the application across browser implementations and the quality of the interface.

Security has been decomposed into *Authorization*, *Confidentiality* and *External Consistency*. Clients, exposed to the internet are, like servers, at risk in web applications. It is possible for web browsers and application servers to download or upload content and programs that could open up the client system to crackers and automated agents. JavaScript, Java applets, ActiveX controls, and plug-ins all represent a certain degree of risk to the system and the information it manages. Equally important, are the procedures checking the consistency of data transactions.

Eventually, the analysis shown in Figure 8 allows us to choose the joint venture architectural style for our e-commerce example (the operationalized attribute is marked with a "$\surd$"). More details about the selection and non-functional requirements decomposition process can be found in [32,33]. In addition, more specific attributes have been identified during the decomposition process, such as *Integrity* (*Accuracy*, *Completeness*), *Usability*, *Response Time*, *Maintainability*, *Updatability*, *Confidentiality*, *Authorization* (*Identification*, *Authentication*, *Validation), Consistency* and need to be considered in the system architecture.

Figure 9 suggests a possible assignment of system responsibilities, based on the joint venture architectural style. The system is decomposed into three

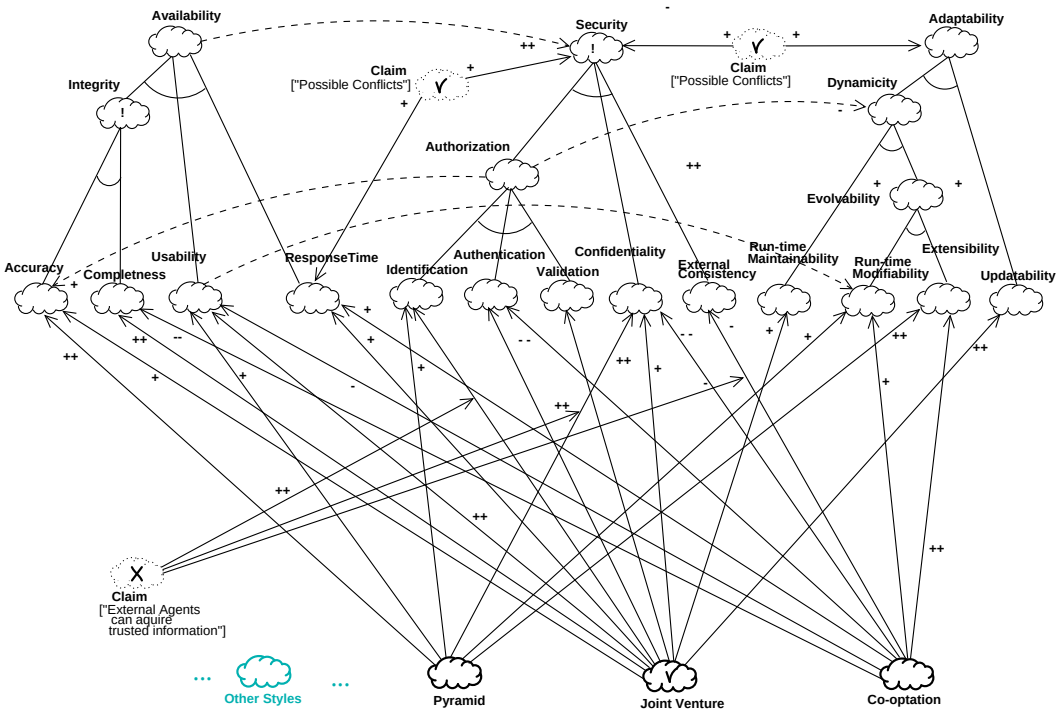

Fig. 8. Selecting an Architecture

principal partners (*Store Front*, *Billing Processor* and *Back Store*) controlling themselves on a local dimension and exchanging, providing and receiving services, data and resources with each other.

Each of them delegates authority to and is controlled and coordinated by the joint management actor (*Joint Manager*) managing the system on a global dimension. *Store Front* interacts primarily with *Customer* and provides her with a usable front-end web application. *Back Store* keeps track of all web information about customers, products, sales, bills and other data of strategic importance to *Media Shop*. *Billing Processor* is in charge of the secure management of orders and bills, and other financial data; also of interactions to *Bank Cpy*. *Joint Manager* manages all of them controlling *security* gaps, *availability* bottlenecks and *adaptability* issues. All four sub-actors need to communicate and collaborate in running the system. For instance, *Store Front* communicates to *Billing Processor* relevant customer information required to process bills. *Back Store* organizes, stores and backs up all information coming from *Store Front* and *Billing Processor* in order to produce statistical analyses, historical charts and marketing data.

In the following, we further detail *Store Front*. This actor is in charge of catalogue browsing and item database searching, also provides on-line customers with detailed information about media items. We assume that different media shops working with *Medi@* may want to provide their customers with various forms of information retrieval (boolean, keyword, thesaurus, lexicon, full text,

indexed list, simple browsing, hypertext browsing, SQL queries, etc.).

Store Front is also responsible for supplying a customer with a web shopping cart to keep track of items the customer is buying when visiting *Medi@*. We assume that different media shops using the *Medi@* system may want to provide customers with different kinds of shopping carts with respect to their internet browser, plug-ins configuration or platform or simply personal wishes (e.g., Java mode shopping cart, simple browser shopping cart, frame-based shopping cart, CGI shopping cart, enhanced CGI shopping cart, shockwave-based shopping cart, ...)

Finally, *Store Front* also initializes the kind of processing that will be done (by *Billing Processor*) for a given order (phone/fax, internet standard form or secure encrypted form). We assume that different media shop managers using the *Medi@* web system may be processing various types of orders, such as those listed above differently and that customers may be selecting the kind of delivery system they would like to use (UPS, FedEx, DHL, express mail, normal mail, overseas service, ...).

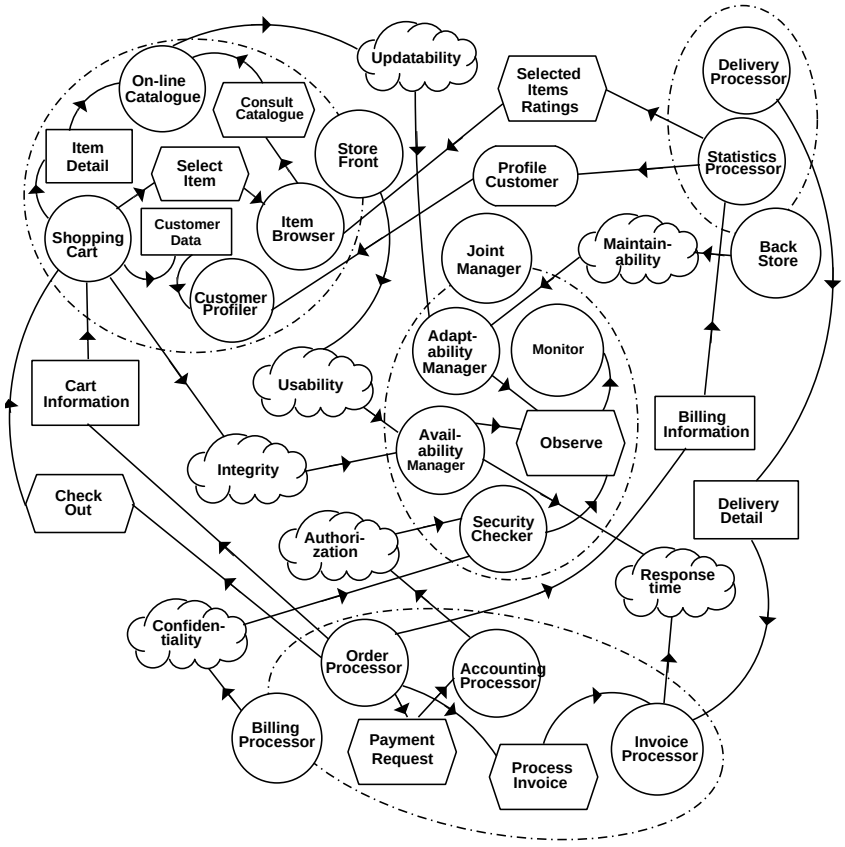

Fig. 9. The E-commerce System as Joint Venture Architecture

Fulfillment of an actor's obligations can be accomplished through delegation

and through decomposition of the actor into component actors. The introduction of other actors described in the previous paragraphs amounts to a form of delegation in the sense that *Store Front* retains its obligations, but delegates subtasks, sub-goals etc. to other actors – an alternative architectural design would have *Store Front* outsourcing some of its responsibilities to some other actors, so that *Store Front* removes itself from the critical path of obligation fulfilment. In addition, as shown in Figure 9, *StoreFront* – and the other system actors – is also refined into an aggregate of actors which, by design, work together to fulfil *Store Front*'s obligations. This is analogous to a committee being refined into a collection of members who collectively fulfil the committee's mandate.

Hence, to accommodate the responsibilities of *Store Front*, we introduce subactors *Item Browser* to manage catalogue navigation, *Shopping Cart* to select and custom items, *Customer Profiler* to track customer data and produce client profiles, and *On-line Catalogue* to deal with digital library obligations.

Moreover, to cope with the identified software quality attributes (*Security*, *Availability* and *Adaptability*), *Joint Manager* is further refined into four new system sub-actors *Availability Manager*, *Security Checker* and *Adaptability Manager* each of them assuming one of the main softgoals (and their more specific subgoals) and observed by a *Monitor*.

Billing Processor is decomposed into *Order Processor* to dialogue with *Shopping Cart* and deals with billing details, *Accounting Processor* to interact with *Bank Cpy* (not represented in Figure 9) and *Invoice Processor* to deal with delivery and invoice information. Finally, *Back Store* is refined into *Statistics Processor* producing charts, reports, audits, sales, turnover forecast, and *Delivery Processor* to interact with information systems of delivery companies.

A further step in the architectural design consists in defining how the goals assigned to each actor are fulfilled by agents with respect to social patterns. For this end, designers can be guided by a catalogue of agent patterns in which a set of pre-defined solutions are available. A lot of work has been done in software engineering for defining software patterns, and many of them, such as those identified in [22,40], can be incorporated into agent system architectures. Unfortunately, they focus on object-oriented not on the inherent intentional and social characteristics of agents.

In the area of multi-agent systems, some work has been done in designing agent patterns, see for instance [1,16,30]. However, these contributions focus on problems like how agents communicate one another, get information from information sources, and establish a connection with a specific host. Differently, in *Tropos*, social patterns are used for solving a specific goal defined at the architectural level through the identification of organizational styles and

relevant quality attributes (softgoals) as discussed previously.

We have defined a catalogue involving some social pattern recurrent in multi-agent literature; in particular, some of the federated patterns introduced in [25,47]: *broker*, *matchmaker*, *mediator*, *monitor*, *embassy*, *wrapper*, *contract-net*.

For instance, a *matchmaker* (Figure 10) locates a provider corresponding to a consumer request for service, and then gives the consumer a handle to the chosen provider directly. Contrary to the broker who directly handles all interactions between the consumer and the provider, the negotiation for service and actual service provision are separated into two distinct phases. This pattern can be used in horizontal integrations and joint ventures [20].

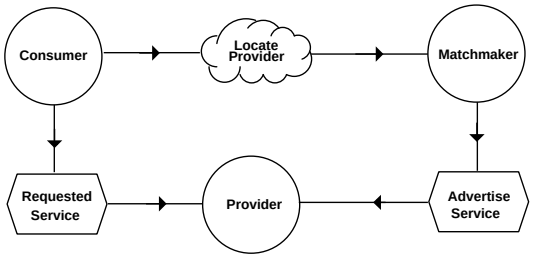

Fig. 10. Matchmacker

An *embassy* (Figure 11) routes a service requested by an foreign agent to a local one and handle back the response. If the access is granted, the foreign agent can submit messages to the embassy for translation. The content is translated in accordance with a standard ontology. Translated messages are forwarded to target local agents. The results of the query are passed back out to the foreign agent, translated in reverse. This pattern can be used in the structure-in-5, arm's-length, bidding and co-optation styles to take in charge security aspects between system components related to the competitivity mechanisms inherent to these styles [20].

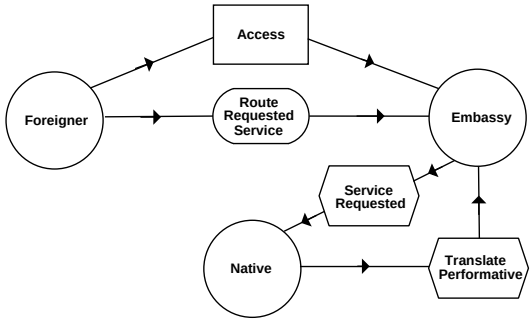

Fig. 11. Embassy

A detailed analysis of each social pattern allows to define a set of capabilities associated with the agents involved in the pattern. Such capabilities are not exhaustive and concern exclusively the agent activities relative to the pattern's goal. Table 2 presents a set of capabilities for the matchmaker pattern.

Table 2

Agents' capabilities for the matchmaker pattern

MATCHMAKER	
Agent	**Capabilities**
Customer	Build a request to query the matchmaker Handle with a services ontology Query the matchmaker for a service Find alternative matchmakers Request a service to a provider Manage possible provider failures Monitor the provider's ongoing processes Ask the provider to stop the requested service
Provider	Handle with a services ontology Advertise a service to the matchmaker Withdraw the advertisement Use an agenda for managing the requests Inform the customer of the acceptance of the request service Inform the customer of a service failure Inform the customer of success of a service
Matchmaker	Update the local database Handle with a services ontology Use an agenda for managing the customer requests Search the name of an agent for a service Inform the customer of the unavailability of agents for a service

A capability states that an actor is able to act in order to achieve a given goal. In particular, for each capability the actor has a set of plans that may apply in different situations. A plan describes the sequence of actions to perform and the conditions under which the plan is applicable. It is important to notice that we have common capabilities for different actors; for instance, the capability "handle services ontology" is common to the three actors of the *Matchmaker* pattern. Capabilities are collected in a catalogue and associated to the pattern. This allows to define the actors' role and capabilities suitable for a particular domain.

Figure 12 shows a possible use of patterns in the e-business system depicted in Figure 9. In particular, it describes how to solve the goal of *managing catalogue navigation* that *Store Front* has delegated to *Item Browser*. The goal is decomposed into different subgoals and solved with a combination of patterns. The broker pattern is applied to *Info Searcher*, which satisfies requests of searching information by accessing *On-line Catalogue*. *Source Matchmaker* applies the matchmaker pattern locating the appropriate source for *Info Searcher*, and the monitor pattern is used to check any possible change in the *On-line Catalogue*. Finally, the mediator pattern is applied to mediate the interaction

among *Info Searcher*, *Source Matchmaker*, and *Wrapper*, while the wrapper pattern makes the interaction between *Item Browser* and *On-line Catalogue* possible. Of course, other patterns can be applied [33]. For instance, we could use the contract-net pattern to select a wrapper to which delegate the interaction with *On-line Catalogue*, or the embassy to route the request of a wrapper to *On-line Catalogue*.

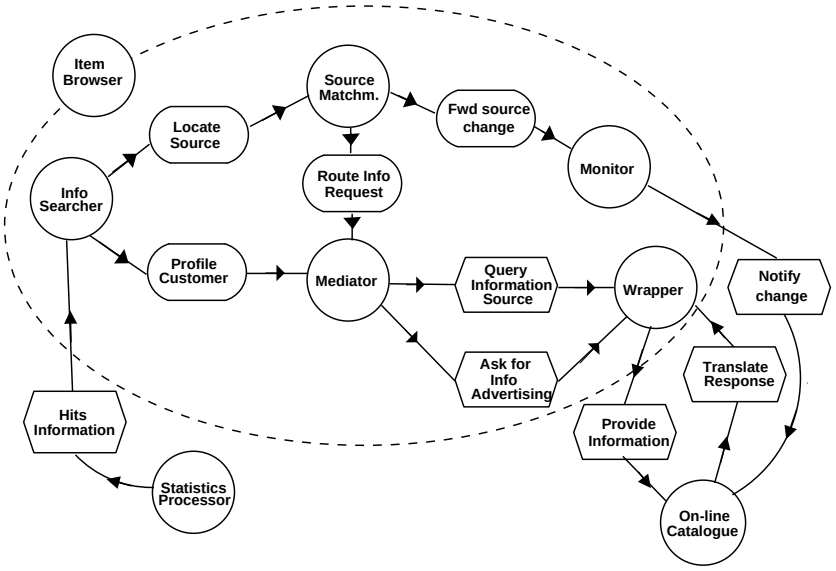

Fig. 12. Detailing Item Browser with Social Patterns

6 Detailed Design

The detailed design phase is intended to introduce additional detail for each architectural component of a system. In our case, this includes actor communication and actor behavior. To support this phase, we propose to adopt existing agent communication languages like FIPA-ACL [35] or KQML [18], message transportation mechanisms and other concepts and tools. One possibility is to adopt extensions to UML [4], like AUML, the Agent Unified Modeling Language [3,38] proposed by the FIPA (Foundation for Physical Intelligent Agents)[19] and the OMG Agent Work group.

We have also proposed and defined a set of stereotypes, tagged values, and constraints to accommodate *Tropos* concepts within UML [37]. As an example, Figure 13 depicts the *i** strategic dependency model from Figure 12 in UML using the stereotypes we have defined, notably $\ll$ *i* actor* $\gg$ and $\ll$ *i* dependency* $\gg$. Such mapping in UML could also be done in a similar way for strategic rationale or goal analysis models.

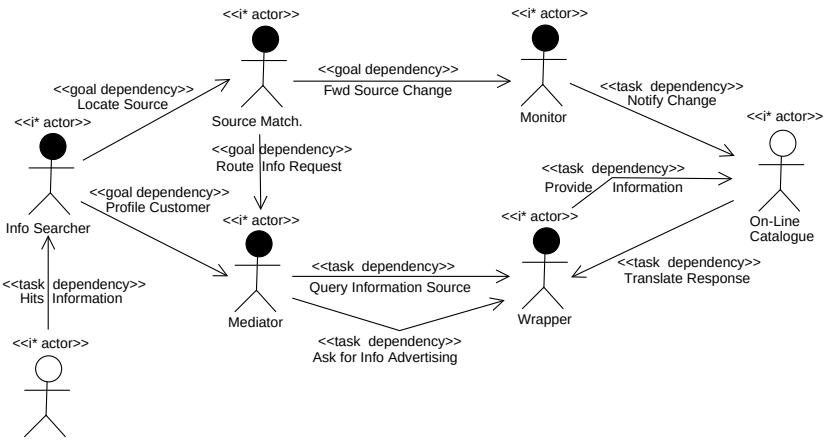

Fig. 13. Representing the i^* Model from Figure 12 in UML with Stereotypes

To illustrate the use of AUML, the rest of the section concentrates on the *Shopping cart* actor and the *check out* dependency. Figure 14 depicts a partial UML class diagram focusing on that actor that will be implemented as an aggregation of several *CartForm*s and *ItemLine*s. Associations *ItemDetail* to *On-line Catalogue*, aggregation of *MediaItem*s, and *CustomerDetail* to *CustomerProfiler*, aggregation of *CustomerProfileCard*s are directly derived from resource dependencies with the same name in Figure 9. Our target implementation model is the BDI model, an agent model whose main concepts are *Beliefs*, *Desires* and *Intentions*. As indicated in Figure 18, we propose to implement i^* tasks as BDI intentions (or plans). We represent them as methods (see Figure 14) following the label "*Plans:*".

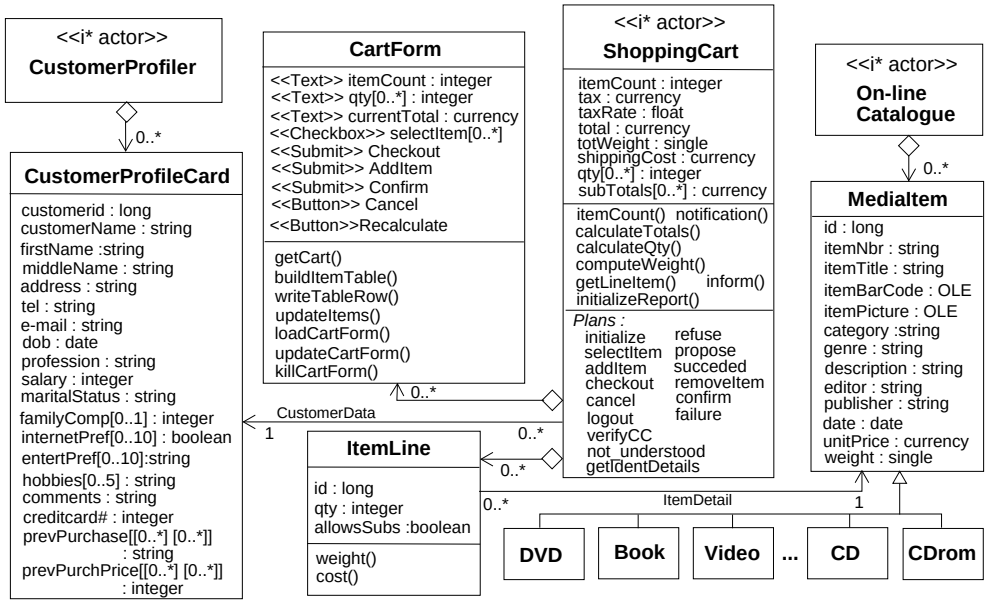

Fig. 14. Partial Class Diagram for *Store Front* Focusing on *Shopping Cart*

To specify the *checkout* task, AUML allows us to use templates and packages to represent *checkout* as an object, but also in terms of sequence and collaborations diagrams.

Figure 15a introduces the *checkout* interaction context which is triggered by the *checkout* communication act (CA) and ends with a returned information status. When the *Customer* pushes the *checkout* button, the *Shopping Cart* asks the *Order Processor* to process orders. In turn, the latter sends a *payment request* CA to *Accouting Processor* which *informs* him about the status (failure/success) of its internal processing. In case of success, *Order Processor* concurrently asks *Invoice Processor* to process the invoice (and send a *delivery detail* CA to *Delivery Processor*) and sends *billing information* to *Statistics Processor*.

This diagram only provides basic specification for an intra-agent order processing protocol. In particular, the diagram stipulates neither the procedure used by the *Customer* to produce the *checkout* CA, nor the procedure employed by the *Shopping Cart* to respond to the CA.

As shown by Figure 15b, such details can be provided by using *levelling* [38], i.e., by introducing additional interaction and other diagrams. Each additional level can express *inter-actor* or *intra-actor* dialogues. At the lowest level, specification of an actor requires spelling out the detailed processing that takes place within the actor. Figure 15b focuses on the protocol between *Customer* and *Shopping Cart* which consists of a customization of the FIPA Contract Net protocol [38]. Such a protocol describes a communication pattern among actors, as well as constraints on the contents of the messages they exchange. When a *Customer* wants to check out, a request-for-proposal message is sent to *Shopping Cart*, which must respond before a given timeout (for network security and integrity reasons). The response may refuse to provide a proposal, submit a proposal, or express miscomprehension. The diamond symbol with an "$\times$" indicates an "exclusive or" decision. If a proposal is offered, *Customer* has a choice of either accepting or canceling the proposal. The internal processing of *Shopping Cart*'s *checkout* plan is described in Figure 16.

At the lowest level, we use plan diagrams [31], to specify the internal processing of atomic actors. Each identified plan is specified as a plan diagram, which is denoted by a rectangular box. The lower section, the plan graph, is a state transition diagram. However, plan graphs are not just descriptions of system behavior developed during design. Rather, they are directly executable prescriptions of how a BDI agent should behave (execute identified plans) to achieve a goal or respond to an event.

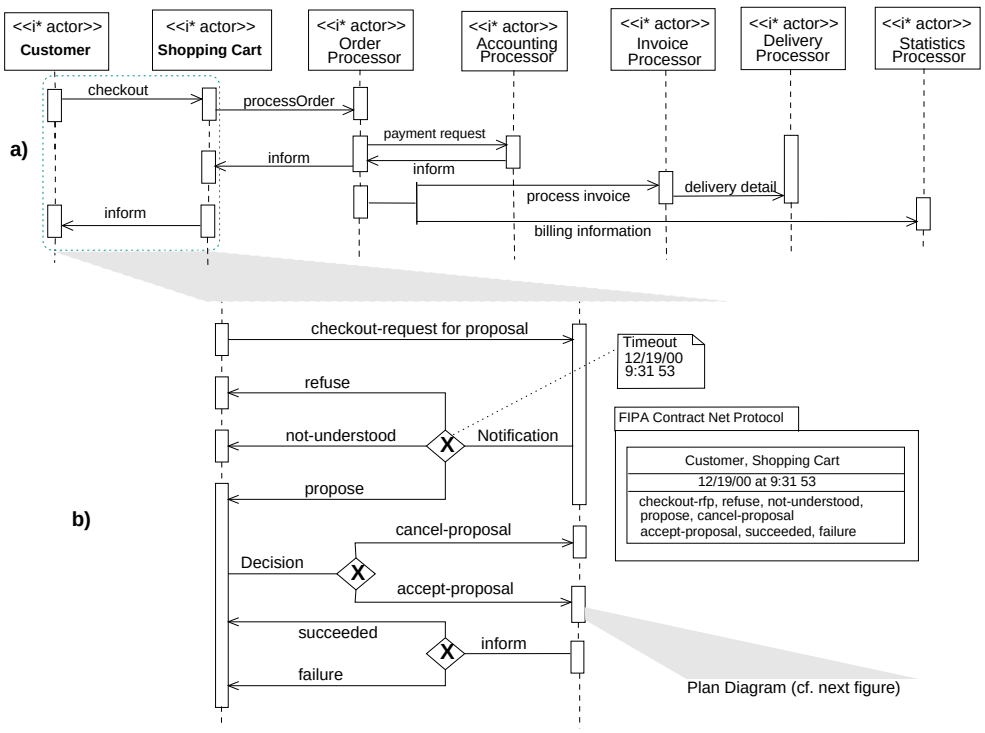

Fig. 15. Sequence Diagram to Order Media Items (a), and Agent Interaction Protocol Focusing on a *Checkout* Dialogue (b)

The initial transition of the plan diagram is labeled with an activation event (*Press checkout button*) and activation condition (*[checkout button activated]*) which determine when and in what context the plan should be activated. Transitions from a state automatically occur when exiting the state and no event is associated (e.g., when exiting *Fields Checking*) or when the associated event occurs (e.g., *Press cancel button*), provided in all cases that the associated condition is true (e.g., *[Mandatory fields filled]*). When the transition occurs any associated action is performed (e.g., *verifyCC()*).

The elements of the plan graph are three types of node; start states, end states and internal states, and one type of directed edge; transitions. Start states are denoted by small filled circles. End states may be pass or fail states, denoted respectively by a small target or a small no entry sign. Internal states may be passive or active. Passive states have no substructure and are denoted by a small open circle. Active states have an associated activity and are denoted by rectangular boxes with rounded corners. An important feature of plan diagrams is their notion of failure. Failure can occur when an action upon a transition fails, when an explicit transition to a fail state occurs, or when the activity of an active state terminates in failure and no outgoing transition is enabled.

Figure 16 depicts the plan diagram for *checkout*, triggered by pushing the checkout button. Mandatory fields are first checked. If any mandatory fields

are not filled, an iteration allows the customer to update them. For security reasons, the loop exits after 5 tries ([i<5]) and causes the plan to fail. Credit Card validity is then checked. Again for security reasons, when not valid, the CC# can only be corrected 3 times. Otherwise, the plan terminates in failure. The customer is then asked to confirm the CC# to allow item registration. If the CC# is not confirmed, the plan fails. Otherwise, the plan continues: each item is iteratively registered, final amounts are calculated, stock records and customer profiles are updated and a report is displayed. When finally the whole plan succeeds, the *Shopping Cart* automatically logs out and asks the *Order Processor* to initialize the order. When, for any reason, the plan fails, the *Shopping Cart* automatically logs out. At anytime, if the cancel button is pressed, or the timeout is more than 90 seconds (e.g., due to a network bottleneck), the plan fails and the *Shopping Cart* is reinitialized.

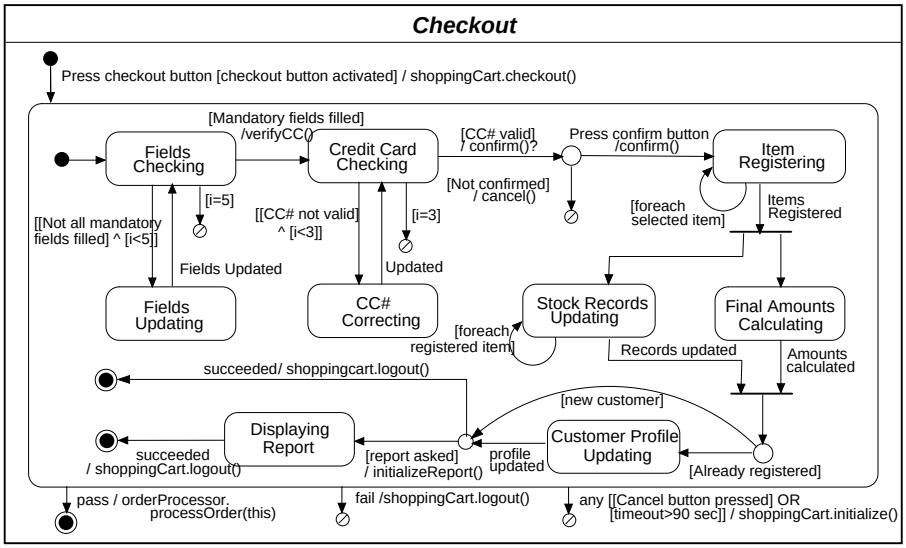

Fig. 16. A Plan Diagram for *Checkout*

Dynamics specifications such as processes modeled by plan diagrams in *Tropos* can be formalized using ConGolog [36,44]. Primitive actions can be defined in terms of pre- and post-conditions and decomposed into procedures using modeling constructs like sequencing ($a_1; a_2$), conditional (*if–then*), iteration (*while <condition> do*), concurrent activities ($a_1||a_2$), priority ($a_1\rangle\rangle a_2$), non-deterministic choice ($a_1|a_2$), interrupt ($< x : \phi \rightarrow \sigma >$ where x is a list of variables, ϕ a trigger condition and σ a body), ... In addition to offering programming language-like structures for describing processes ConGolog underlying logic is designed to support reasoning with respect to process specifications and simulations. Figure 17 gives some ConGolog specifications for the *checkout* plan graph of Figure 16.

Proc *checkOutShoppingCart(shopCart)*
$< shopCart : failed(shopCart) \rightarrow logoutShoppingCart(shopCart) >$
$\rangle\rangle$
$(< pressedCancelButton \rightarrow reinitializeShoppingCart(shopCart) >$
$||$
$< timeout > 90 \rightarrow reinitializeShoppingCart(shopCart) >)$
$\rangle\rangle$
$shopCart : ActivatedCheckoutButton \wedge PressedCheckoutButton$
$\rightarrow startCheckOut(shopCart) >$
EndProc

Fig. 17. ConGolog-like specification for the *checkout* plan from Figure 16

7 Generating an Implementation

JACK Intelligent Agents [10] is an agent-oriented development environment designed to extend Java with the theoretical *Belief Desire Intention* (BDI) agent model [5] used in artificial intelligence as well as in cognitive science and philosophy.

JACK agents can be considered autonomous software components that have explicit *goals* to achieve, or *events* to cope with (desires). To describe how they should go about achieving these desires, agents are programmed with a set of plans (intentions). Each plan describes how to achieve a goal under different circumstances. Set to work, the agent pursues its given goals (desires), adopting the appropriate plans (intentions) according to its current set of data (beliefs) about the state of the world.

To support the programming of BDI agents, JACK offers five principal language constructs: *agents*, *capabilities*, *database relations*, *events*, and *plans*. *Capabilities* aggregate events, plans, databases or other capabilities, each of them assuming a specific function attached to an agent. *Database relations* store beliefs and data of an agent. *Events* identify the circumstances and messages that an agent can respond to. *Plans* are instructions an agent follows to achieve its goals and handle its designated events.

Figure 18 summarizes the mapping from i^* concepts to JACK constructs and how each concept is related to the others within the same model. i^* actors, (informational/data) resources, softgoals, goals and tasks are respectively mapped into BDI agents, beliefs, desires and intentions. In turn, a BDI agent will be mapped as a JACK agent, a belief will be asserted (or retracted) as a database relation, a desire will be posted (sent internally) as a BDIGoalEvent (representing an objective that an agent wishes to achieve), and handled as a plan and an intention will be implemented as a plan. Finally, an i^* dependency will be directly realized as a BDIMessageEvent (received by agents from other agents).

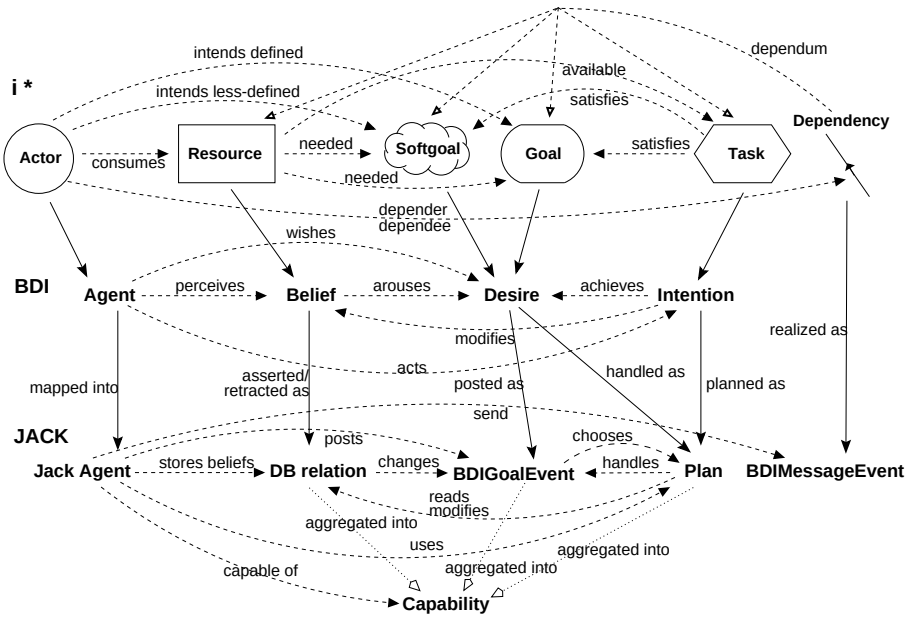

Fig. 18. *i**/BDI/JACK mapping overview

Figure 19 depicts the JACK layout presenting each of the five JACK constructs as well as the implementation of the first part of the dialogue shown in Figure 15b. The request for proposal *checkout-rfp* is a MessageEvent (*extends MessageEvent*) sent by *Customer* and handled by the *Shopping Cart*'s *checkout* plan (*extends Plan*). *Customer* and *Shopping Cart* are implemented as JACK agents (*extends Agent*). In response to *checkout-rfp*, *Shopping Cart* posts a *notification* MessageEvent handled by (one of the) three plans *refuse*, *propose*, *not-understood*. Finally, *Timeout* (which we consider a belief) is implemented as a closed world (i.e., true or false) database relation asserting for each *Shopping Cart* one or several timeout delays.

8 Conclusion and Discussion

We have proposed a development methodology named *Tropos*, founded on intentional and social concepts, and inspired by early requirements analysis. The modeling framework views software from five complementary perspectives:

- **Social** – who are the relevant actors, what do they want? What are their obligations? What are their capabilities?
- **Intentional** – what are the relevant goals and how do they interrelate? How are they being met, and by whom ask dependencies?
- **Communicational** – how the actors dialogue and how can they interact with each other?

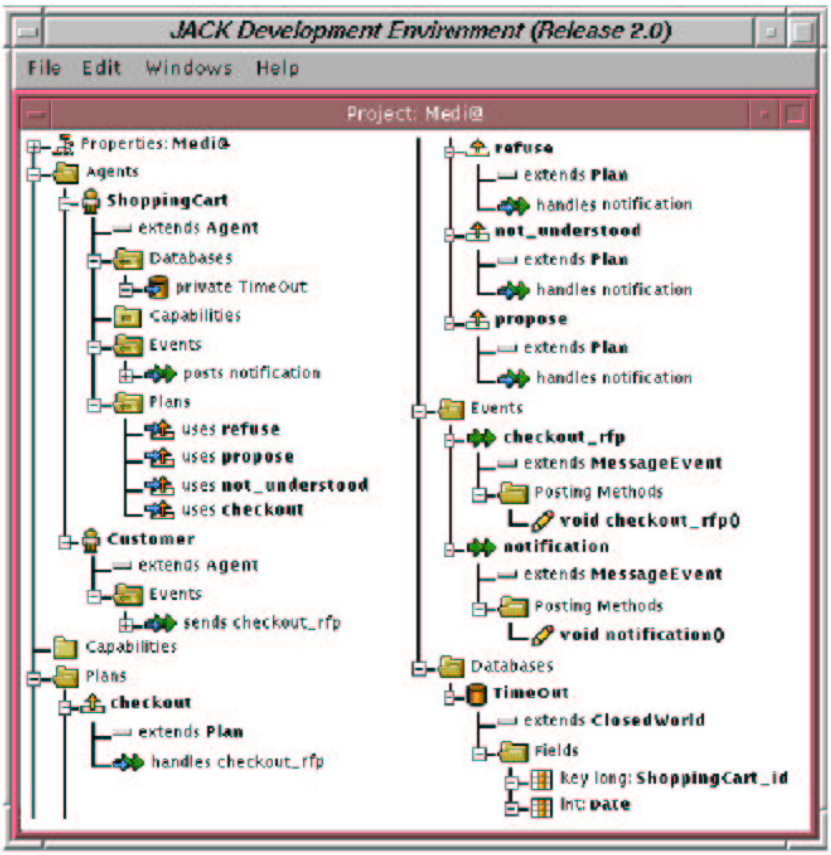

Fig. 19. Partial Implementation of Figure 15b in JACK

- **Process-oriented** – what are the relevant business/computer processes? Who is responsible for what?
- **Object-oriented** – what are the relevant objects and classes, along with their inter-relationships?

We believe that the methodology is particularly appropriate for generic, componentized systems like e-business applications that can be downloaded and used in a variety of operating environments and computing platforms around the world. Preliminary results (e.g., [33,39]) suggest that the methodology complements well proposals for agent-oriented programming environments given that the software is defined in terms of (system) actors, goals and social dependencies among them and that we do not necessarily operationalize these intentional and social structures early on during the development process, avoiding to freeze solutions to a given requirement in the produced software designs.

There already exist some proposals for agent-oriented software development like [3,27,28,31,41,46,48]. Such proposals are mostly extensions to known object-oriented and/or knowledge engineering methodologies. Moreover, all these proposals focus on design – as opposed to requirements analysis – and are therefore considerably narrower in scope than *Tropos*. Indeed, *Tropos* proposes to

adopt the same concepts, inspired by requirements modeling research, for describing requirements *and* system design models in order to narrow the semantic gap between them. The architecture and software design models produced within our framework are intentional in the sense that system components have associated goals that are supposed to be fulfilled. They are also social in the sense that each component has obligations/expectations towards/from other components. Obviously, such models are best suited to cooperative, dynamic and distributed applications like multi-agent systems.

The research reported here is still in progress. Much remains to be done to further refine the proposed methodology and validate its usefulness with real case studies. We are currently working on the development of additional formal analysis techniques for *Tropos* including temporal analysis (using model-checking), goal analysis and social structures analysis, also the development of tools which support different phases of the methodology and the definition of the *Formal Tropos* language.

Acknowledgements

We are greatful to our colleagues Eric Yu and Ariel Fuxman (University of Toronto), also Yves Lespérance (York University, Canada), Fausto Giunchiglia, Paolo Giorgini, Anna Perini and Paolo Bresciani (University of Trento and IRST) for their contributions to the *Tropos* project.

This project has been partially funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada, also by the Province of Ontario through CITO, a centre of excellence for research on Communications and Information Technology.

This work was carried out while Jaelson Castro and Manuel Kolp were visiting the Department of Computer Science, University of Toronto.

References

[1] Y. Aridor and D. Lange. Agent design patterns: Elements of agent application design. In *Proc. of the 2nd Int. Conf. on Autonomous Agents, Agents'98*, pages 108–115, St. Paul, USA, May 1998.

[2] S. Baiman. Agency research in managerial accounting: a second look. *Accounting, Organizations and Society*, 15(4):341–371, 1990.

[3] B. Bauer, J. Muller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction. In *Proc. of the 1st Int. Workshop on Agent-Oriented Software Engineering, AOSE'00*, pages 91–104, Limerick, Ireland, 2001.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language: User Guide*. Addison-Wesley, 1999.

[5] M. Bratman. *Intention, plans, and practical reason*. Harvard University Press, 1987.

[6] J. Castro, M. Kolp, and J. Mylopoulos. Developing agent-oriented information systems for the enterprise. In B. Sharp, editor, *Enterprise Information Systems II*. Kluwer Publishing, 2001.

[7] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proc. of the 13th Int. Conf. on Advanced Information Systems Engineering, CAiSE'01*, pages 108–123, Interlaken, Switzerland, June 2001.

[8] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] M. Coburn. Jack intelligent agents: User guide version 2.0. At http://www.agent-software.com, 2001.

[11] P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 32(3):213–261, 1990.

[12] J. Conallen. *Building Web Applications with UML*. Addison-Wesley, 2000.

[13] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, 1993.

[14] A. Davis. *Software Requirements: Objects, Functions and States*. Prentice Hall, 1993.

[15] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.

[16] D. Deugo, F. Oppacher, J. Kuester, and I. Otte. Patterns as a means for intelligent software engineering. In *Proc. of the Int. Conf. of Artificial Intelligence, ICAI'01*, pages 605–611, Las Vegas, USA, July 1999.

[17] Y. Doz and G. Hamel. *Alliance Advantage: The art of creating value through partnership*. Harvard Business School Press, 1998.

[18] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[19] FIPA. The Foundation for Intelligent Physical Agents. At http://www.fipa.org, 2001.

[20] A. Fuxman, P. Giorgini, M. Kolp, and J. Mylopoulos. Information systems as social structures. In *Proc. of the 2nd Int. Conf. on Formal Ontologies for Information Systems, FOIS'01*, Ogunquit, USA, Oct. 2001.

[21] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specification in Tropos. In *Proc. of the 5th Int. Symposium on Requirements Engineering, RE'01*, Toronto, Canada, Aug. 2001.

[22] E. Gamma, R. Helm, J. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[23] P. Giorgini, M. Kolp, and J. Mylopoulos. Multi-agent architectures as organizational structures. Submitted to *Journal of Autonomous Agents and Multi Agent Systems*, 2001.

[24] GRL. Goal oriented requirement language. At http://www.cs.toronto.edu/km/GRL, 2001.

[25] S. Hayden, C. Carrick, and Q. Yang. Architectural design patterns for multiagent coordination. In *Proc. of the 3rd Int. Conf. on Autonomous Agents, Agents'99*, Seattle, USA, May 1999.

[26] IBM. Patterns for e-business. At http://www.ibm.com/developerworks/patterns, 2001.

[27] C. Iglesias, M. Garrijo, and J. Gonzalez. A survey of agent-oriented methodologies. In *Proc. of the 5th Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages, ATAL'98*, pages 317–330, Paris, France, Oct. 1999.

[28] N. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

[29] M. Jensen and W. Meckling. Theory of the firm: Managerial behaviour, agency costs and capital structure. *Journal of Financial Economics*, 3(2):305–360, 1976.

[30] E. Kendall, P. M. Krishna, C. Pathak, and C. Suersh. Patterns of intelligent and mobile agents. In *Proc. of the 2nd Int. Conf. on Autonomous Agents, Agents'98*, pages 92–99, St. Paul, USA, May 1998.

[31] D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In *Proc. of the 3rd Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages, ATAL'96*, pages 1–20, Budapest, Hungary, Aug. 1997.

[32] M. Kolp, J. Castro, and J. Mylopoulos. A social organization perspective on software architectures. In *Proc. of the 1st Int. Workshop From Software Requirements to Architectures, STRAW'01*, pages 5–12, Toronto, Canada, May 2001.

[33] M. Kolp, P. Giorgini, and J. Mylopoulos. A goal-based organizational perspective on multi-agents architectures. In *Proc. of the 8th Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages, ATAL'01*, Seattle, USA, Aug. 2001.

[34] M. Kolp and J. Mylopoulos. Software architectures as organizational structures. In *Proc. ASERC Workshop on "The Role of Software Architectures in the Construction, Evolution, and Reuse of Software Systems"*, Edmonton, Canada, Aug. 2001.

[35] Y. Labrou, T. Finin, and Y. Peng. The current landscape of agent communication languages. *Intelligent Systems*, 14(2):45–52, 1999.

[36] Y. Lespérance, T. Kelley, J. Mylopoulos, and E. Yu. Modeling dynamic domains with ConGolog. In *Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering CAiSE'99*, pages 108–123, Heidelberg, Germany, June 1999.

[37] J. Mylopoulos, M. Kolp, and J. Castro. UML for agent-oriented software development: The Tropos proposal. In *Proc. of the 4th Int. Conf. on the Unified Modeling Language UML'01*, Toronto, Canada, Oct. 2001.

[38] J. Odell, H. Van Dyke Parunak, and B. Bauer. Extending UML for agents. In *Proc. of the 2nd Int. Bi-Conference Workshop on Agent-Oriented Information Systems, AOIS'00*, pages 3–17, Austin, USA, July 2000.

[39] A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. In *Proc. of the 5th Int. Conf on Autonomous Agents, Agents'01*, Montreal, Canada, May 2001.

[40] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[41] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. Van de Velde, and B. Wielinga. *Knowledge engineering and management: the CommonKADS methodology*. MIT Press, 2000.

[42] W. Scott. *Organizations: rational, natural, and open systems*. Prentice Hall, 1998.

[43] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[44] X. Wang and Y. Lespérance. Agent-oriented requirements engineering using ConGolog and i*. In *Proc. of the 3rd Int. Bi-Conference Workshop on Agent-Oriented Information Systems, AOIS'01*, Montreal, Canada, May 2001.

[45] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[46] M. Wood and S. DeLoach. An overview of the multiagent systems engineering methodology. In *Proc. of the 1st Int. Workshop on Agent-Oriented Software Engineering, AOSE'00*, pages 207–222, Limerick, Ireland, 2001.

[47] S. Woods and M. Barbacci. Architectural evaluation of collaborative agent-based systems. Technical Report CMU/SEI-99-TR-025, SEI, Carnegie Mellon University, Pittsburgh, USA, 1999, 1999.

[48] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[49] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.

[50] E. Yu. Modeling organizations for information systems requirements engineering. In *Proc. of the 1st Int. Symposium on Requirements Engineering, RE'93*, pages 34–41, San Jose, USA, Jan. 1993.

[51] E. Yu. Understanding 'why' in software process modeling, analysis and design. In *Proc. of the 16th Int. Conf. on Software Engineering, ICSE'94*, pages 159–168, Sorrento, Italy, May 1994.

[52] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, 1995.

[53] E. Yu and J. Mylopoulos. Using goals, rules, and methods to support reasoning in business process reengineering. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 5(1):1–13, 1996.

A Outline of the *Tropos* Methodology

(1) **Acquisition of Early Requirements**. The outputs of this phase are two models.
   (a) Strategic Dependency (SD) Model to capture relevant actors, theirs respective goals and their interdependencies.
   (b) Strategic Rationale (SR) Model to determine through a means-end analysis how the goals can be fulfilled through the contributions of other actors.

(2) **Definition of Late Requirements in *i**.** The outputs of this phase are revised SD and SR models.
   (a) Include in the original Strategic Dependency (SD) Model an actor to represent the software system to be developed.
   (b) Take this system actor and do a means-ends analysis to produce a new Strategic Rational (SR) Model.
   (c) If necessary decompose the system actor into several sub-actors and revise the SD and SR Models.

(3) **Architectural design**. The outputs of this phase are a Non Functional Requirements (NFR) Diagram and revised SD and SR models. Agents are introduced.

(a) Select an architectural style using as criteria the desired qualities identified in Step 2. Produce a NFR diagram to represent the selection and design rationale.
(b) If required, introduce new system actors and dependencies, as well as the decomposition of existing actors and dependencies into sub-actors and sub-dependencies. Revise the SD and SR Models.
(c) Assigning actors to agents and roles/patterns to solve actors'goals.

(4) **Detailed design**. The outputs of this phase are Agent Class Diagrams, Sequence Diagrams, Collaboration Diagrams and Plan Diagrams.

(a) Based on the SD and SR models produce a Class Diagram.
(b) Develop Sequence and Collaboration diagrams to capture inter-actor dynamics,
(c) Develop Plan (state-based) Diagrams to capture both intra-actor and inter-actor dynamics.

(5) **Implementation**. The output of this phase is a BDI (Beliefs-Desires-Intentions) agent architecture.

From the detailed design generate Agents, Capabilities, Database Relations, Events and Plans in JACK.